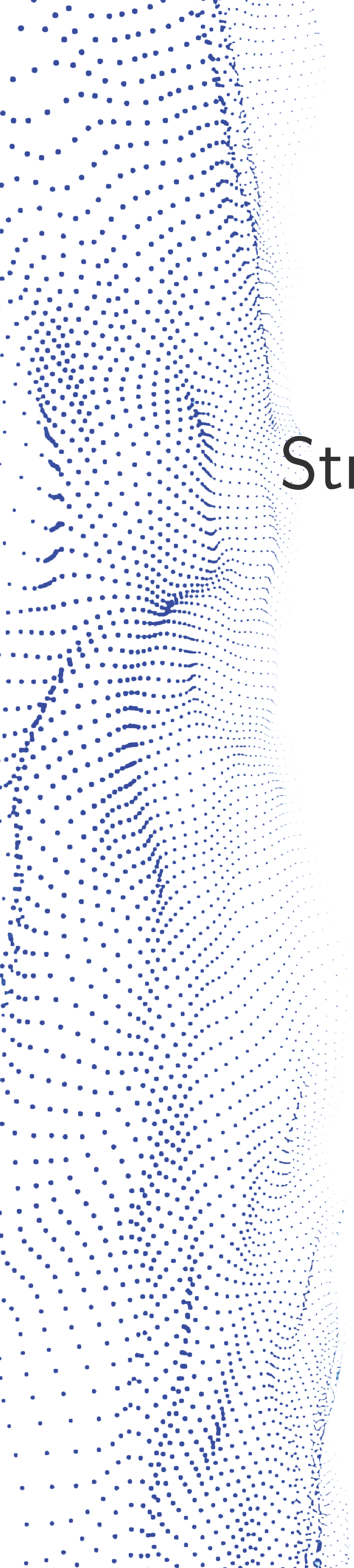


Online Audio-Video Streaming for Interactive Music Performances.

University of Trento

Academic Year 2021/22

Bonomi Andrea, Momesso Filippo, Turri Evelyn

Table of Contents

Abstract

Our project aims at analyzing feasibility and implementing strategies to perform low latency streaming of point cloud data over the internet using Unity and Stereolabs ZED camera. We proposed several solutions trying to reach this goal, each one with its technical issues due primarily to lack of support by the technologies used. In this report we discuss in detail our proposed approaches.

1 Introduction

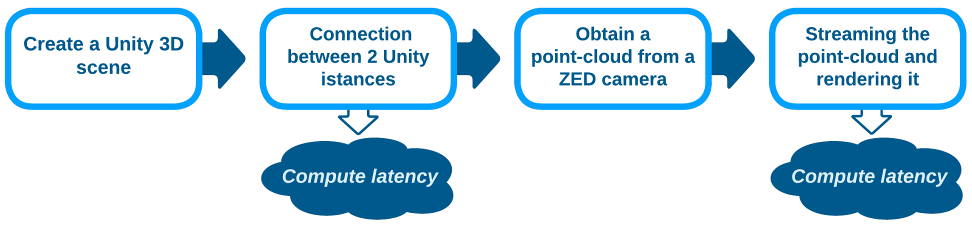
Recently, with the advent of the Metaverse, the ICT industry has been pushed even more towards the development of Virtual Reality applications. During the pandemic, social distancing and other restrictions made it difficult for musicians and performers to continue their job or enjoy jam sessions with others. Furthermore, we have experienced the difficulty in using video call applications as a way to perform live in a social distancing setting, due to low quality audio, low quality video and high latency.
Our project is a component of a bigger picture which aims to allow musicians to connect in virtual reality and perform live, with low latency and high quality 3D video. Our contribution is meant to analyze the feasibility of 3D point cloud streaming in Unity with currently available technologies.

2 Technologies

Our project is developed in Unity and takes advantage of two main technologies: Unity Render Streaming package and ZED stereo camera. Unity Render Streaming is an experimental Unity package which provides Audio, Video and Input controls streaming over the Internet taking advantage of WebRTC protocol. ZED Camera is a stereo camera for depth perception and motion tracking. It is provided with two 4MP camera sensors and has plug & play integrations with various softwares including Unity.
In order to develop our project we used the following hardware and software:

- **Hardware:** Stereolabs ZED Camera, Nvidia GPU compatible computer with CUDA.
- **Software:** Unity 2020.3.8, Unity Render Streaming 3.1.10-exp.3 [1], Unity WebRTC 2.4.0-exp.7 [7], Nvidia CUDA Toolkit 11.5.2 and ZED SDK 3.7.4.

3 Workflow


Create a Unity 3D scene
Connection between 2 Unity istances
Obtain a point-cloud from a ZED camera
Streaming the point-cloud and rendering it
Compute latency
Compute latency

3.1 Develop a 3D Unity scene

The scene can be very simple or very complex, the main idea is to have a camera for viewing the scene, a UI that includes buttons for managing the connection and other settings, i.e. the presence or not of the point cloud in the scene, and lastly the ZED gameObject. The ZED gameObject has to include the ZED Manager script in order to manage a single ZED camera per scene (Unity does not allow two ZED Manager in the same scene) and the ZEDPointCloudManager script in order to render the point cloud in the scene. If we do not use ZED Manager's own streaming plugin we also need to set up the connection through the RenderStreaming component.

3.2 Instantiate a connection between 2 Unity instances

The connection between two Unity instances is based on a P2P network and managed by RenderStreaming. To perform the connection it is used a signaling server that connects two peers (the Sender and the Receiver) and enables the Sender to stream directly to the Receiver Video, Audio or Input data. RenderStreaming takes advantage of WebRTC technology which is based on UDP protocol, thus there is no way to know if there are packet losses or if a packet arrives at the Receiver. Moreover, two necessary components are Single Connection and Broadcast.

The first one lists components that stream media or data via a single peer connection, it is necessary in order to maintain a connection from the sender. The second one lists components that stream media or data via multiple peer connections, it is necessary in order to create the connection with the receiver.
Initially, we needed to instantiate a connection between two Unity instances and stream some video between them to understand how to work with RenderStreaming before working with point clouds. To do so, taking as reference the several samples available in the documentation, we developed a simple scene with two Unity virtual cameras, one to display the UI and one to work as a simulator of the ZED camera. The texture captured by the latter is streamed through a VideoStreamSender component and displayed on a RawImage object in the receiver instance.

3.3 Compute the latency

In order to compute the latency between the two clients it was necessary to send a timestamp information together with the frame from one end-point to the other. Doing so by using RenderStreaming out-of-the-box features was not possible. To temporarily overcome this problem we printed the timestamp in the UI and streamed the scene captured by the virtual camera with the current timestamp. This way, the local timestamp is at the bottom and the remote timestamp in the frame we are receiving. Then, we compute the difference a posteriori from a screen capture of the running apps by extracting the two timestamps with a simple Python script using both OpenCV and pytesseract libraries (Algorithm 1).
An example can be seen in Figure 1a. The latency data we obtained can be viewed in figure 1b. On average, in streaming only video by using standard Render Streaming functionality we got a latency of ~7 ms with the two clients on a WiFi local network.

3.4 Obtain a point-cloud from a ZED camera

In order to interact with the ZED camera in Unity we need the ZED plugin for Unity, which provides an interface to the C++API. The main components we need are ZEDManager, which provides a way to manage the camera, all its settings and various features and ZEDPointCloudManager which provides a way to render the point cloud in the Unity 3D scene.
ZEDPointCloudManager component works by retrieving two Texture2D objects from the ZEDCamera which are then set on a dedicated Material. This Material is rendered in the scene with its associated textures, through a dedicated shader .
The point cloud is encoded by two textures: one with format ARGBFloat containing the XYZ coordinates for each pixel and one with format BGRA32 containing the RGBA color information for each pixel. ARGBFloat format stores each channel as a IEEE 754 32 bit floating point number while BGRA32 format stores each of RGBA color channels as an 8-bit value in [0..1] range. This information will be useful later in this report. [6]

3.5 Streaming and rendering the point-cloud

This step was the main problematic in the project, and was the one that stopped us in achieving the goal. Different approaches have been tried but none of them allowed us to reach the final goal of streaming the point cloud with high quality and low latency. Firstly we tried to work with Unity Render Streaming and Unity WebRTC experimental packages we used in sections 3.2 and 3.3. Secondly we tried the camera streaming feature provided by the ZED SDK which unfortunately works only on local networks and not over the Internet.

Stream double width texture

Starting from the code in CameraStreamSender script of the Render Streaming package we needed a way to stream together the two textures retrieved from the ZEDCamera. To accomplish this we created a RenderTexture with doubled width and same height w.r.t. the textures retrieved from the ZEDCamera and by using Graphics.Blit() copied on the left half the XYZ texture and on the right half the color texture.
The combined RenderTexture needs to be in BGRA32 format, the only one supported by Unity WebRTC package. Due to the incompatibility between ARGBFloat format of the XYZ Texture and BGRA32 format, coordinates values, originally in [-inf, +inf] range, were clipped in [0,1] range during copy. Therefore, through this approach it was impossible to reconstruct and render the point cloud after receiving the streamed texture.

Normalization & denormalization

To avoid the aforementioned clipping, we tried to normalize the XYZ coordinates data in [0,1] range before the copy on the combined RenderTexture. This has been achieved by applying standard normalization on each coordinate using ZEDCamera's negative and positive depth range as interval endpoints, though this is clearly suboptimal. This decision comes from the fact that computing min and max values for each frame was too computationally expensive.

After receiving the streamed combined texture and extracting the two textures for color and XYZ coordinates, the latter is denormalized.
This approach has a fundamental flaw which is that, by normalizing in [0,1] and storing those values in an 8 bit per channel texture, a quantization of the data is applied. Therefore, clusters of points are rendered on the same location giving as a result a spaced-grid effect as we can see in Figure 3. As said, information is lost due to the encoding in 8bit format and not due to normalization and denormalization, as we can see in Figure 2 where no format conversion is applied. Format conversion however is required and cannot be avoided due to Unity WebRTC supporting only BGRA32 format.
In order to try to reconstruct, at least partially, the lost information we tried to apply firstly a gaussian blur to the XYZ coordinates and secondly a bilinear interpolation. The resulting quality however is poor as we can see in Figure 4 and in Figure 5.

Working with bytes

To avoid the loss of information due to format conversion we experienced with previous solutions, we thought that working directly with the raw byte data of the texture would have been a more suitable approach.
As previously said, BGRA32 format stores each pixel's data as 8 bits per channel (4 bytes per pixel), while ARGBFloat format is encoded as a 32 bit floating point number per channel (16 bytes per pixel). Therefore, an ARGBFloat texture takes four times the memory required by a BGRA32 texture. Our idea was to take the raw bytes of the original texture and put them directly in a new BGRA32 RenderTexture with $width_{new} = 4 * width$ which is then copied into the combined texture to be streamed as in previous approaches. In this way 4 pixels of the new texture encode one original (float 32) pixel.
We tested this idea in a local environment without involving any streaming in the process, by only applying the copy of raw bytes data from the original ARGBFloat texture into the new BGRA32 texture and then copying them back again. As a result, no information was lost and the reconstructed point cloud presented good quality. However, when we included the streaming between the two *Unity* instances through Unity WebRTC, the left part of the received texture, which contains the XYZ coordinates, resulted as completely corrupted and the point cloud could not be rendered. Conversely, the right part containing the RGB color information showed no damage.
By inspecting the source code of Unity WebRTC package and thanks to a conversation with its main developer *kazuki_unity729* [9] we found out that this behavior is due to how the textures are processed and encoded before the actual streaming. The actual streamed data is encoded to H.264 by NvCodec (on Windows) which requires frames to be in YUV420 format. Therefore the BGRA32 is first converted into YUV420 and then encoded. The receiver client decodes the H.264 frame into YUV420 format which is then converted back to BGRA32.
The reason why XYZ coordinates result as corrupted in this latter approach can be found in the BGRA32 $\rightarrow$ YUV420 and YUV420 $\rightarrow$ BGRA32 conversions: YUV420 format performs downsampling on the two chrominance values, therefore part of the information is lost and, consequently, leads to a corrupted texture. In our approach each pixel of the streamed texture encodes a byte, in its entirety. In other words, each channel is treated as a [0,1] value even if actually it is not. Therefore any channel-wise computation might modify the underlying 8 bits and if not invertible leads to a completely different float value when the texture is reconstructed back into ARGBFloat format.

Native ZEDCamera streaming on local network

Last experiment was to use the native ZEDCamera streaming on local network [3]. This time the setup is very different from before, we removed everything mentioned previously and developed a simple scene where the user can set the IP address of the sender in order to enable reception on the receiver. This time the resulting quality is very good as we can see in Figure 6. However, there are some problems with this procedure.
First of all the two peers must be on the same network to obtain very good results, it is possible to use a VPN but if the internet connection that is not good enough the resulting latency is very high, i.e. in the order of seconds, likely due to the VPN overhead.
Moreover, latency can not be computed anymore because the ZED Manager streaming is implemented in a compiled DLL, which we can not edit in order to stream the timestamp together with the texture. Another issue due to this closed-source implementation of the native streaming feature is the fact that we can not reduce the size point cloud by downsampling or discarding unnecessary points. As a consequence it is not possible for us to reduce the bandwidth required.

3.6 Latency of point-cloud streaming

To compute latency of point cloud streaming is not possible anymore to exploit the technique presented in 3.3. To overcome this we found a way of encoding the timestamp into a 5×5 texture that we can send together with the combined texture described in 3.5, in an analogous way[1]. This way we avoid printing the timestamp on the UI.

[1]Adapted from the Latency Sample described in [8]

The algorithm is the same as Algorithm 1, the only difference is that the decoder includes an additional step which converts the frame to text and then to timestamp. Moreover, there are no constraints about the dimensions of the encoded frame, thus, we can use the same width and height of the point cloud.
With this approach we measured an average latency of ~170 ms on a local network over WiFi when no post-processing was applied (as described in Section 3.5) while a latency of 1200 ms on the same WiFi network when Gaussian Blur was applied. This higher value is clearly due to the computation of the gaussian blur.

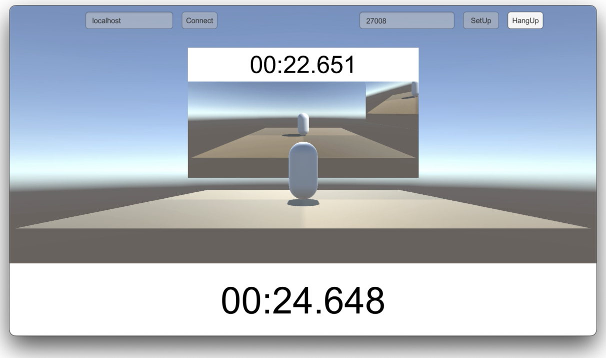
4 Conclusions

We conclude that with the technologies we inspected it is not possible to stream the point cloud without accepting compromises on the quality or latency. The main issue derives from the lack of support in streaming texture with the format used by ZED SDK. Moreover, the other approach we tried allows high quality streaming but lacks flexibility in its customization and works only local networks.

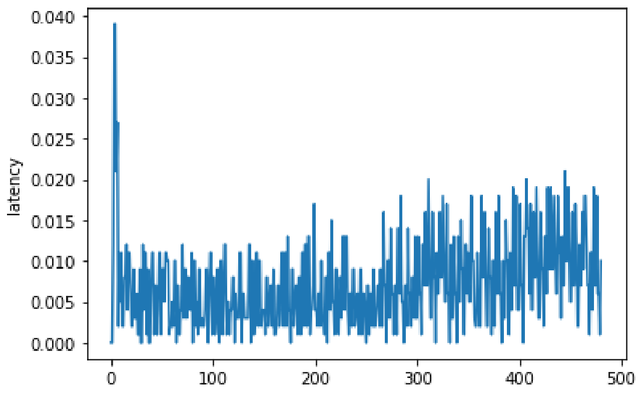
4.1 Future work

Unity WebRTC and RenderStreaming packages receive regular updates and new features thus, in the future, point cloud streaming could be supported. Nonetheless, different software other than Unity, such as Unreal Engine or different capture devices could be considered. Indeed, we found that Microsoft Kinect can overcome the format conversion problems since point coordinates are encoded as 8 bit numbers per channel, therefore it would be easier to deal with. However, Kinect is outperformed by the ZED camera both in quality of the acquired stereo videos and in available features, such as AI powered object detection and motion tracking. Furthermore, Unity developer *kazuki_unity729* suggested that thanks to our contribution the team could consider adding support to YUV444 lossless format in the WebRTC package [9]. Hopefully, the third approach we explored in Section 4 would work, even though there is the possibility that H.264 encoding and decoding steps could still perturb the data making it impossible to reconstruct the original XYZ coordinates. Other solutions we did not investigate include GStreamer which is a framework used to create custom media pipelines and can be used to perform H.264 streaming over UDP [2].

A Appendix



(a) Frame Example for Latency Computation



(b) Latency

Figure 1: Latency information of a streaming video

Algorithm 1 Latency Script

```
function $Latency\text{-}Extraction(recording)$ return a list of latency values
    $latency =$ an empty array
    for each $frame$ in $recording$ do
        Convert $frame$ to text                    ▷ We get the timestamps in string format
        Convert the string timestamps to $timestamp_{local}$ and $timestamp_{remote}$
        if $timestamp_{local}$ and $timestamp_{remote}$ are setted for the first time then
            $first_{local} = timestamp_{local}$
            $first_{remote} = timestamp_{remote}$
        $local = timestamp_{local} - first_{local}$
        $remote = timestamp_{remote} - first_{remote}$
        $latency.add(remote - local)$
    return latency
```

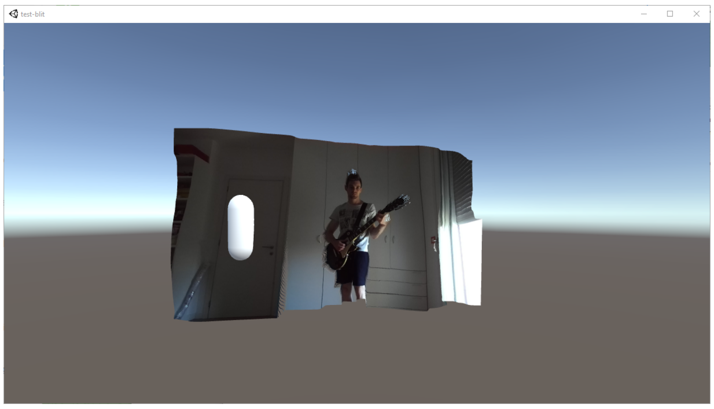


Figure 2: Point-cloud rendering rendering after normalization and denormaliztion without streaming.

In all the figures below the left side shows the sender view while the right shows the receiver view.

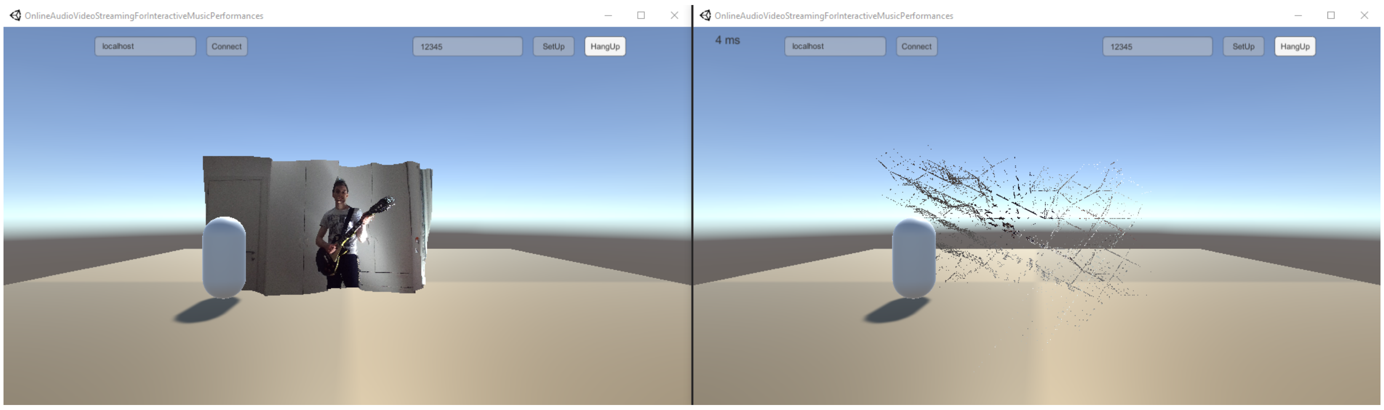

OnlineAudioVideoStreamingForInteractiveMusicPerformances
localhost
Connect
12345
SetUp
HangUp
OnlineAudioVideoStreamingForInteractiveMusicPerformances
4 ms
localhost
Connect
12345
SetUp
HangUp

Figure 3: Point-cloud rendering with no processing.

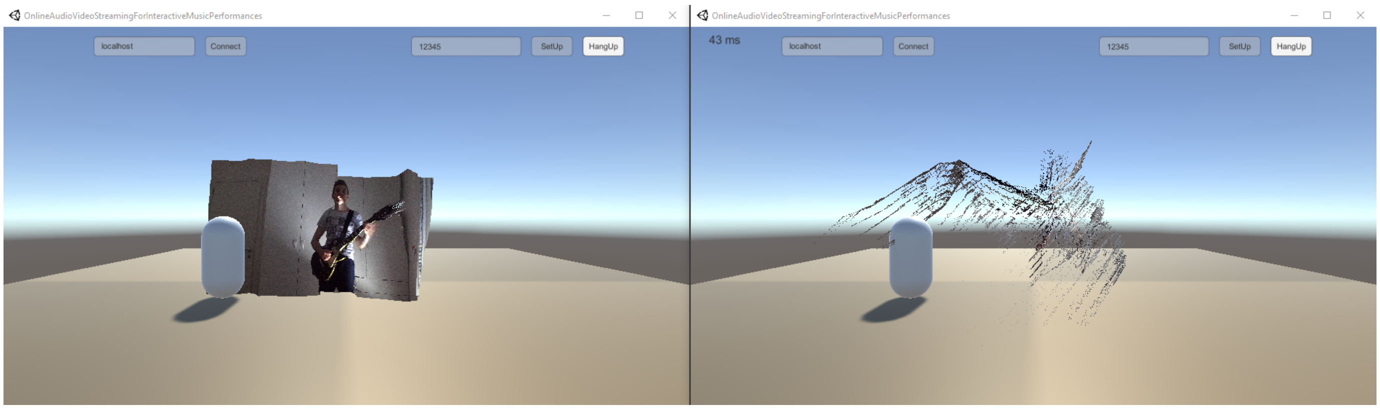

OnlineAudioVideoStreamingForInteractiveMusicPerformances
localhost
Connect
12345
SetUp
HangUp
OnlineAudioVideoStreamingForInteractiveMusicPerformances
43 ms
localhost
Connect
12345
SetUp
HangUp

Figure 4: Point-cloud rendering applying a gaussian blur.

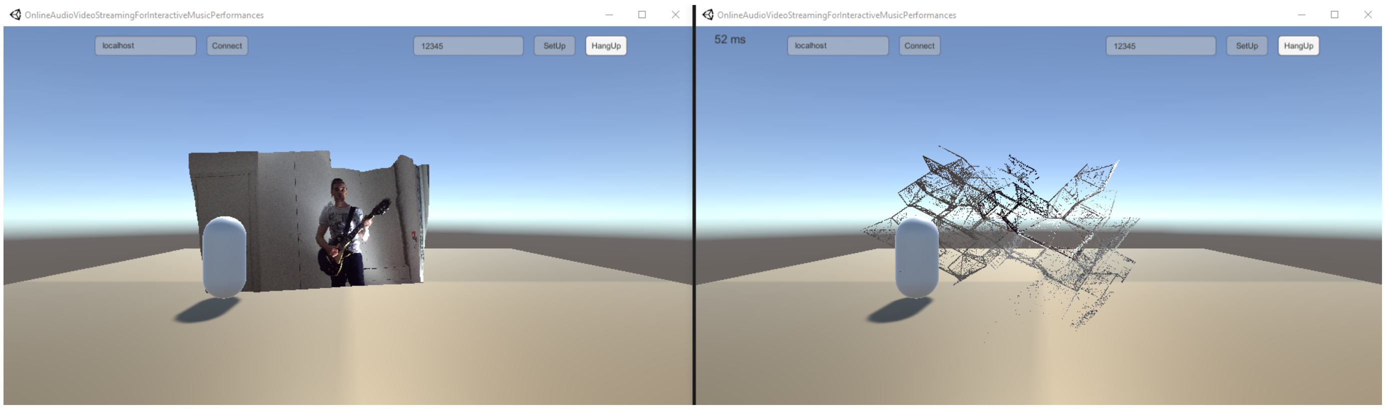

OnlineAudioVideoStreamingForInteractiveMusicPerformances
localhost
Connect
12345
SetUp
HangUp
OnlineAudioVideoStreamingForInteractiveMusicPerformances
52 ms
localhost
Connect
12345
SetUp
HangUp

Figure 5: Point-cloud rendering applying bilinear interpolation.

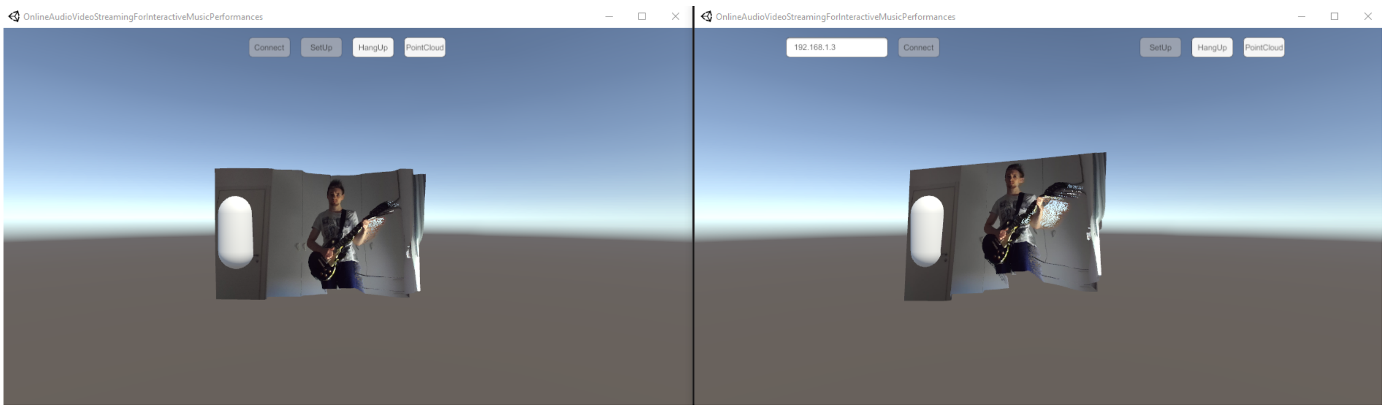

OnlineAudioVideoStreamingForInteractiveMusicPerformances
Connect
SetUp
HangUp
PointCloud
OnlineAudioVideoStreamingForInteractiveMusicPerformances
192.168.1.3
Connect
SetUp
HangUp
PointCloud

Figure 6: Native ZEDCamera streaming.

References

[1] *About Unity Render Streaming*. 2022. URL: https://docs.unity3d.com/Packages/com.unity.renderstreaming@3.1/manual/index.html.

[2] *GStreamer*. 2021. URL: https://www.stereolabs.com/docs/gstreamer/.

[3] *StereoLabs - Local Video Streaming*. 2021. URL: https://www.stereolabs.com/docs/video/streaming/.

[4] *StereoLabs Documentation*. 2022. URL: https://www.stereolabs.com/docs/unity/.

[5] *Stereolabs ZED - Unity plugin*. 2022. URL: https://github.com/stereolabs/zed-unity.

[6] *Unity - TextureFormat*. 2022. URL: https://docs.unity3d.com/ScriptReference/TextureFormat.html.

[7] *WebRTC*. 2022. URL: https://docs.unity3d.com/Packages/com.unity.webrtc@2.4/manual/index.html.

[8] *WebRTC - Samples*. 2022. URL: https://docs.unity3d.com/Packages/com.unity.webrtc@2.4/manual/sample.html.

[9] *WebRTC support for float format textures/point cloud*. Jun 23, 2022. URL: https://forum.unity.com/threads/webrtc-support-for-float-format-textures-point-cloud.1299681/.